

EECS 370 - Lecture 14

Pipelining and Data Hazards II



Live Poll + Q&A: [slido.com #eeecs370](https://slido.com/#eeecs370)

Poll and Q&A Link

Announcements

- Lab 7 due Wed
 - Lab 8 meets Fr/M
- P2L due Thurs
- Homework 3 due Mon 11/6
- Midterm Scores
 - Median: 70%
 - Stdev: 20%
 - Course grade thresholds dropped 1%

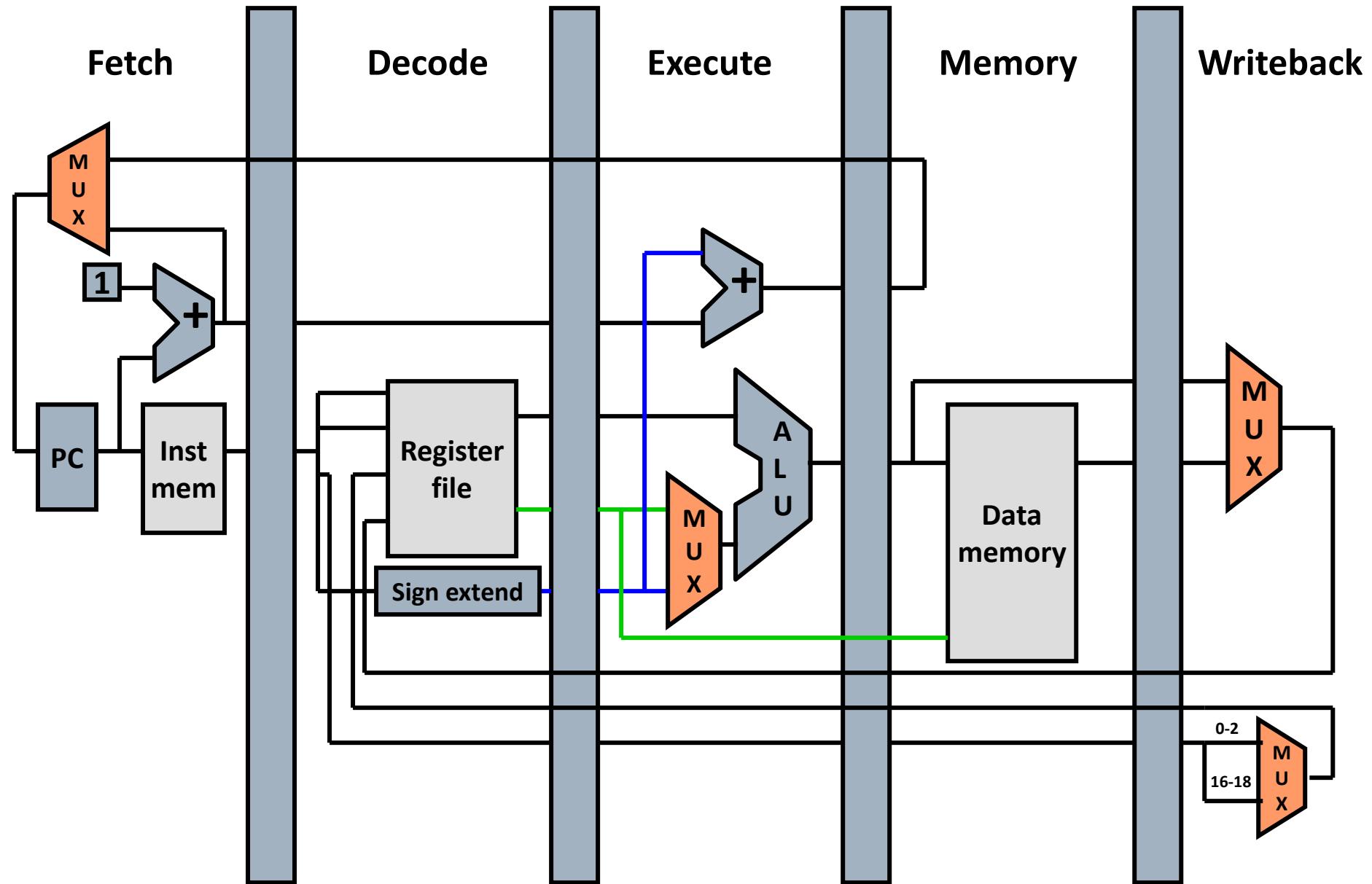


Review: Pipelining

- Goal:
 - Achieve low clock period of multi-cycle processor...
 - ... while maintaining low cycles-per-instruction (CPI) of single cycle processor (close to 1)
 - Achieve this by overlapping execution of multiple instructions simultaneously



Review: Our new pipelined datapath



Review: Sample Code (Simple)

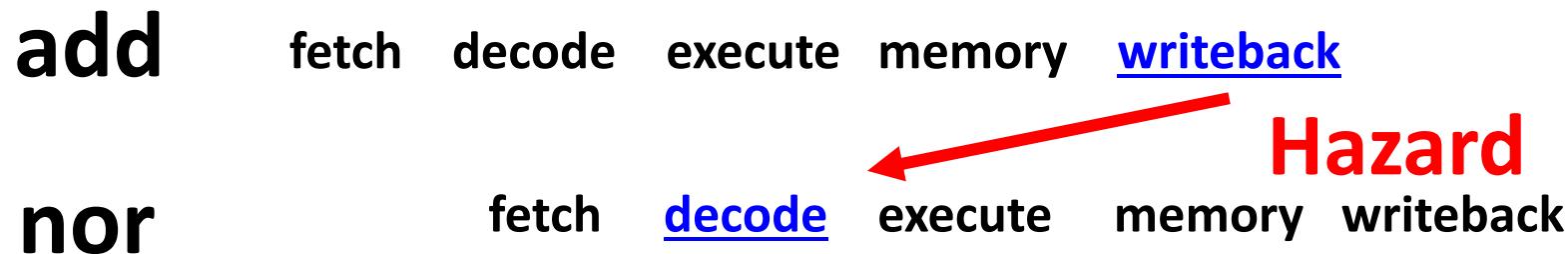
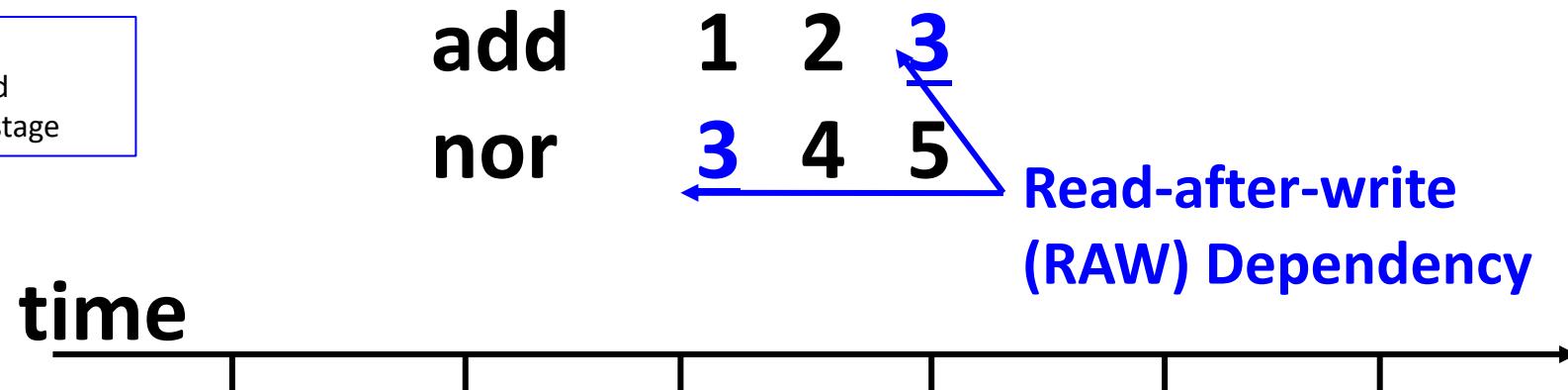
Let's run the following code on pipelined LC2K:

- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



Data Hazards

Recall: registers
are read /sourced
In the “decode” stage



If not careful, nor will read a stale value of register 3

Three approaches to handling data hazards

- Avoid
 - Make sure there are no hazards in the code
- Detect and Stall
 - If hazards exist, stall the processor until they go away.
- Detect and Forward
 - If hazards exist, fix up the pipeline to get the correct value (if possible)



Problems with detect and stall

- CPI increases every time a hazard is detected!
- Is that necessary? Not always!
 - Re-route the result of the **add** to the **nor**
 - **nor** no longer needs to read R3 from reg file
 - It can get the data later (when it is ready)
 - This lets us complete the decode this cycle
 - But we need more control logic



Handling data hazards III: Detect and forward

- Detect: same as detect and stall
 - Except that all 4 hazards have to be treated differently
 - i.e., you can't logical-OR the 4 hazard signals
 - Forward:
 - New **bypass datapaths** route computed data to where it is needed
 - New MUX and control to pick the right data
 - **Beware:** Stalling may still be required even in the presence of forwarding
- since we need to routing them from different locations to different locations.*

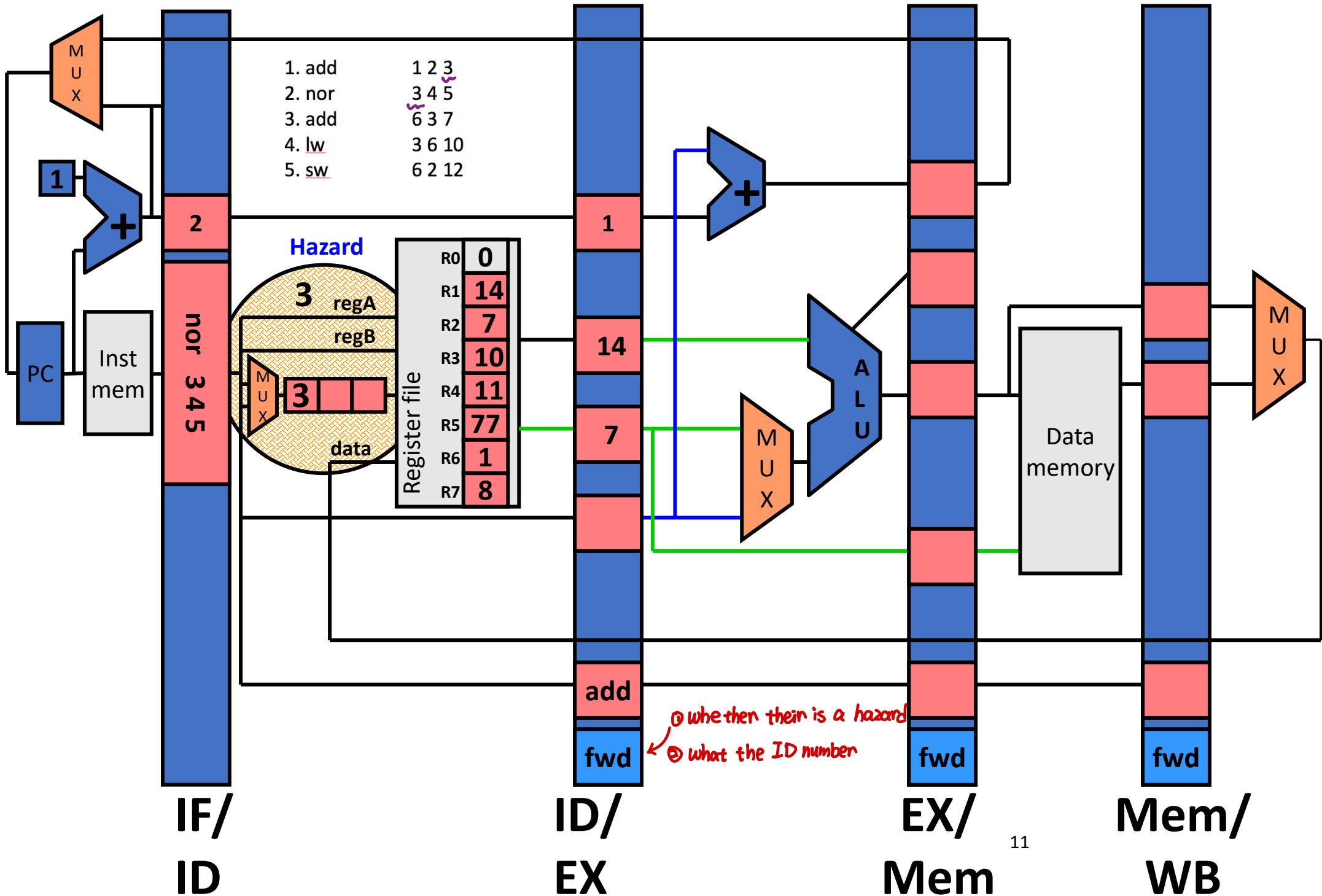
Forwarding example

- We will use this program for the next example
(same as last pipeline diagram example)

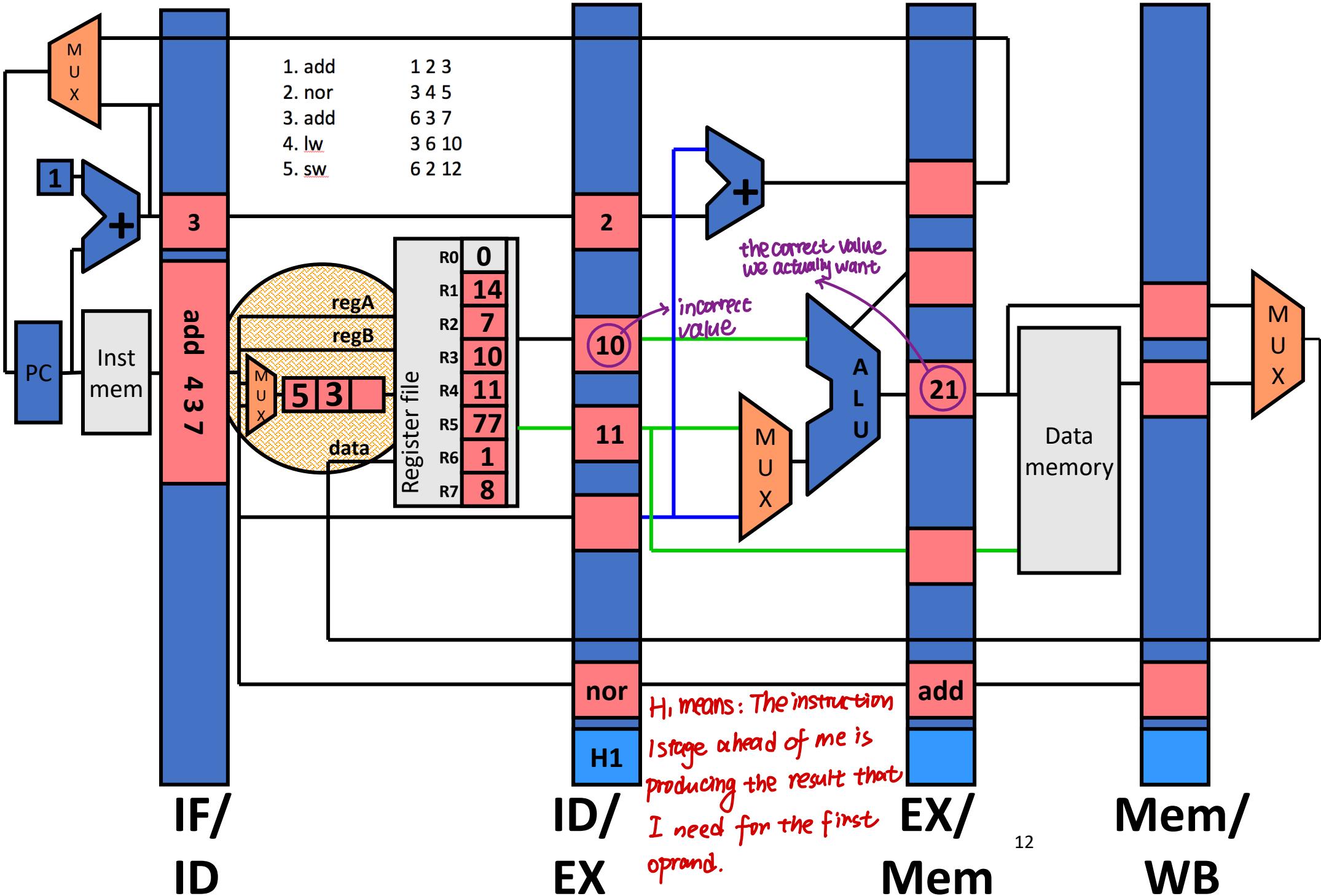
1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12



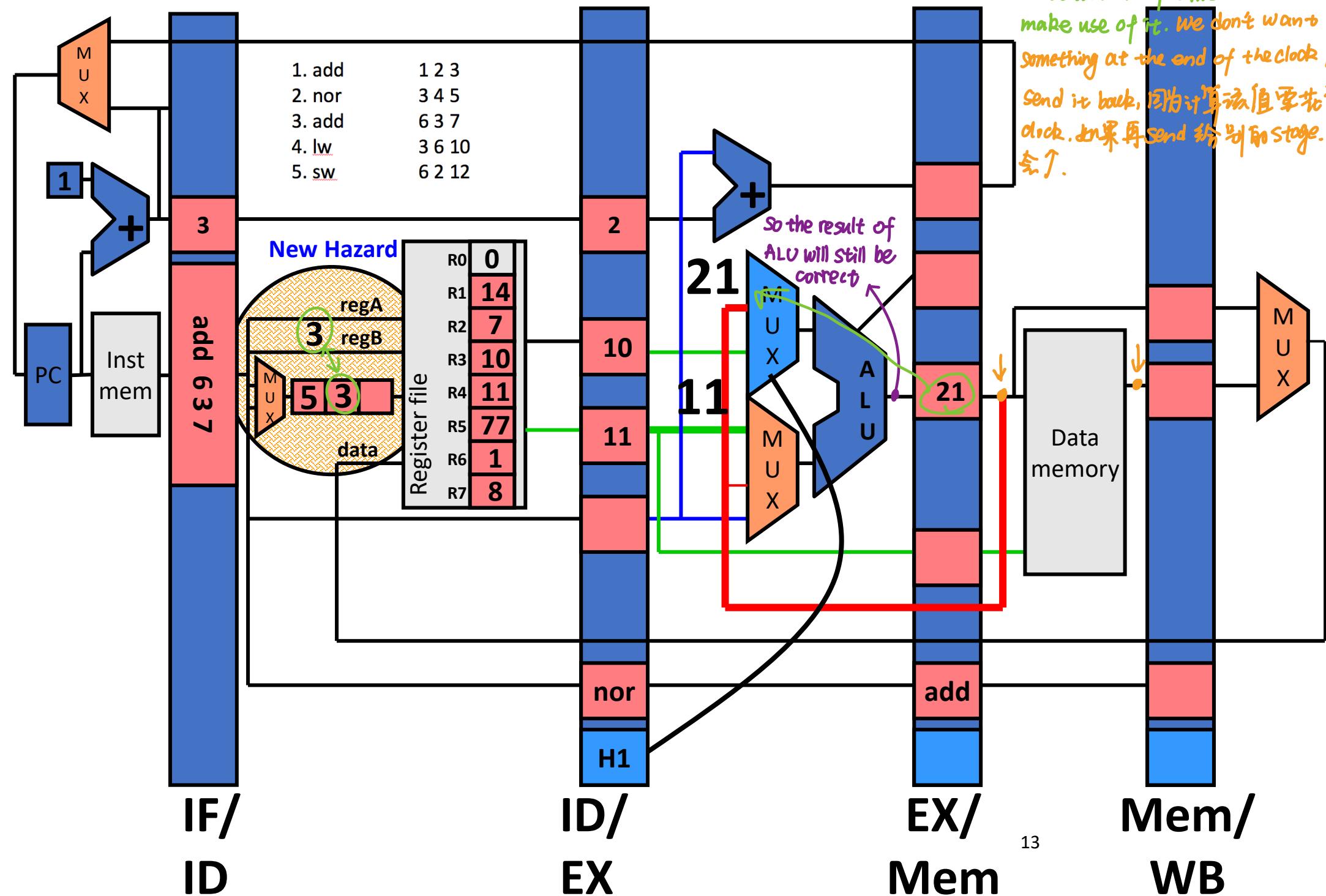
First half of cycle 3



End of cycle 3

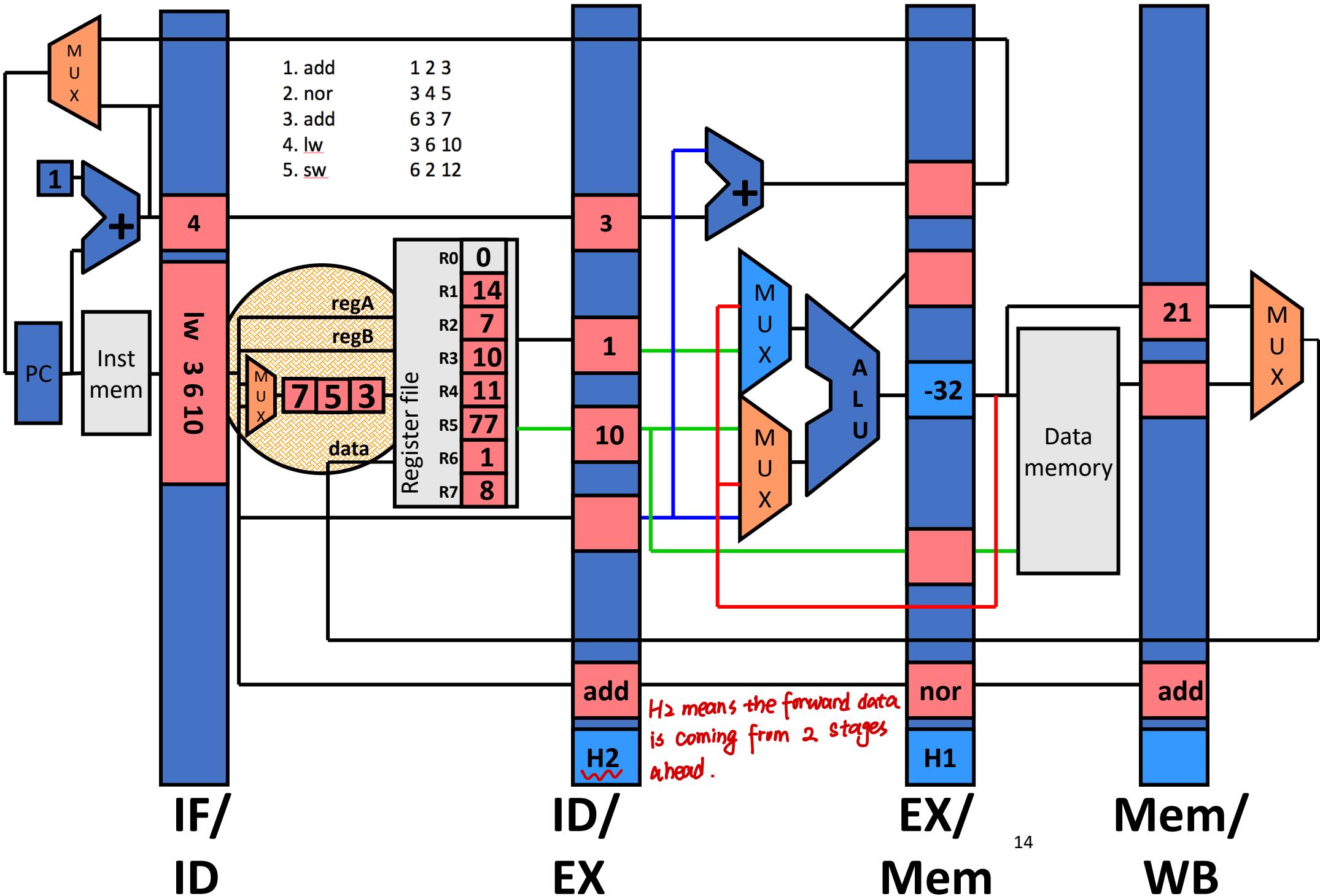


First half of cycle 4

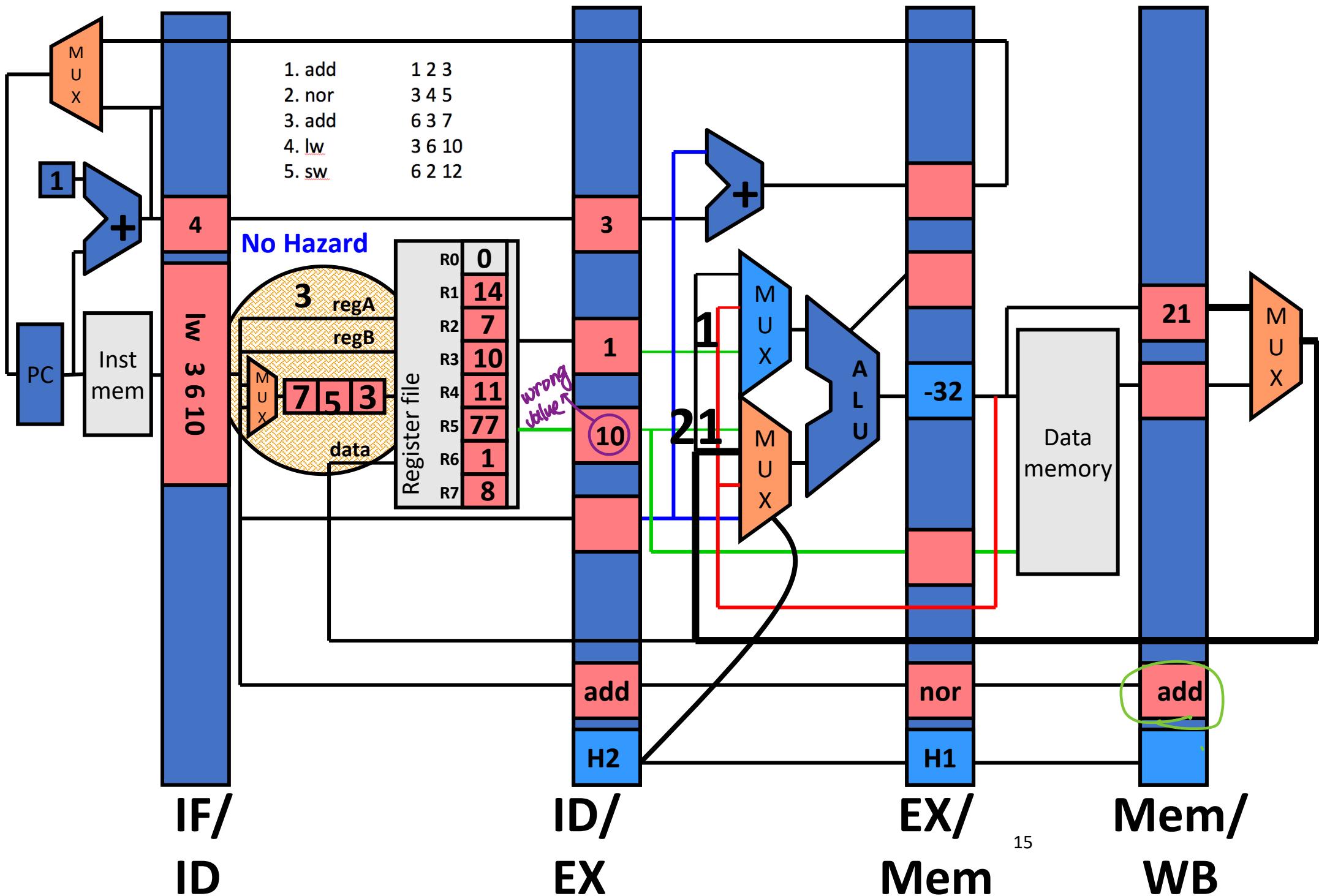


We like to read directly out from the pipeline register, → since we will have the most amount of time in the clock period to make use of it. We don't want to take something at the end of the clock period and send it back, 這樣會浪費 most of clock. 這樣再 send the result to stage. All clock period.

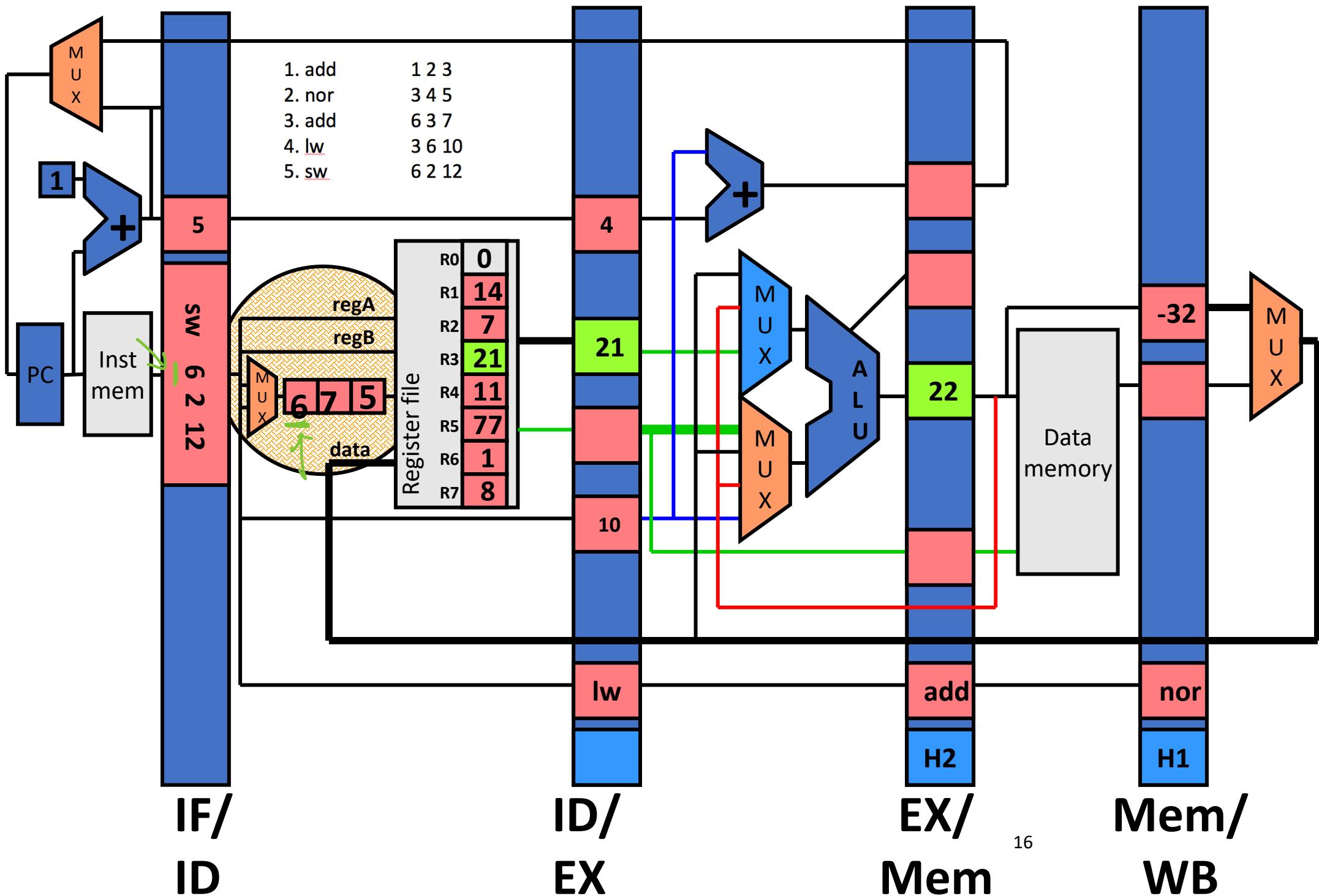
End of cycle 4



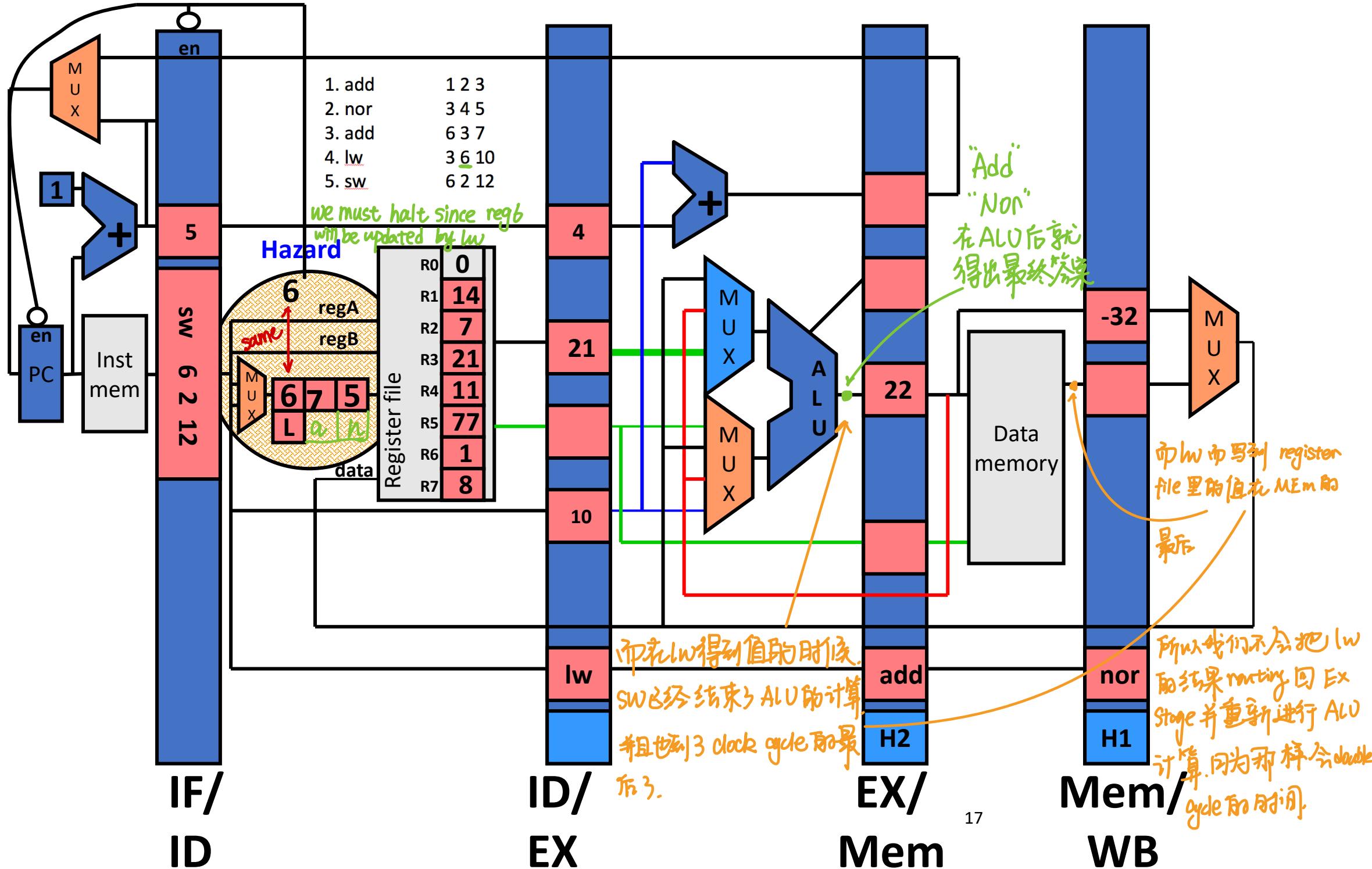
First half of cycle 5



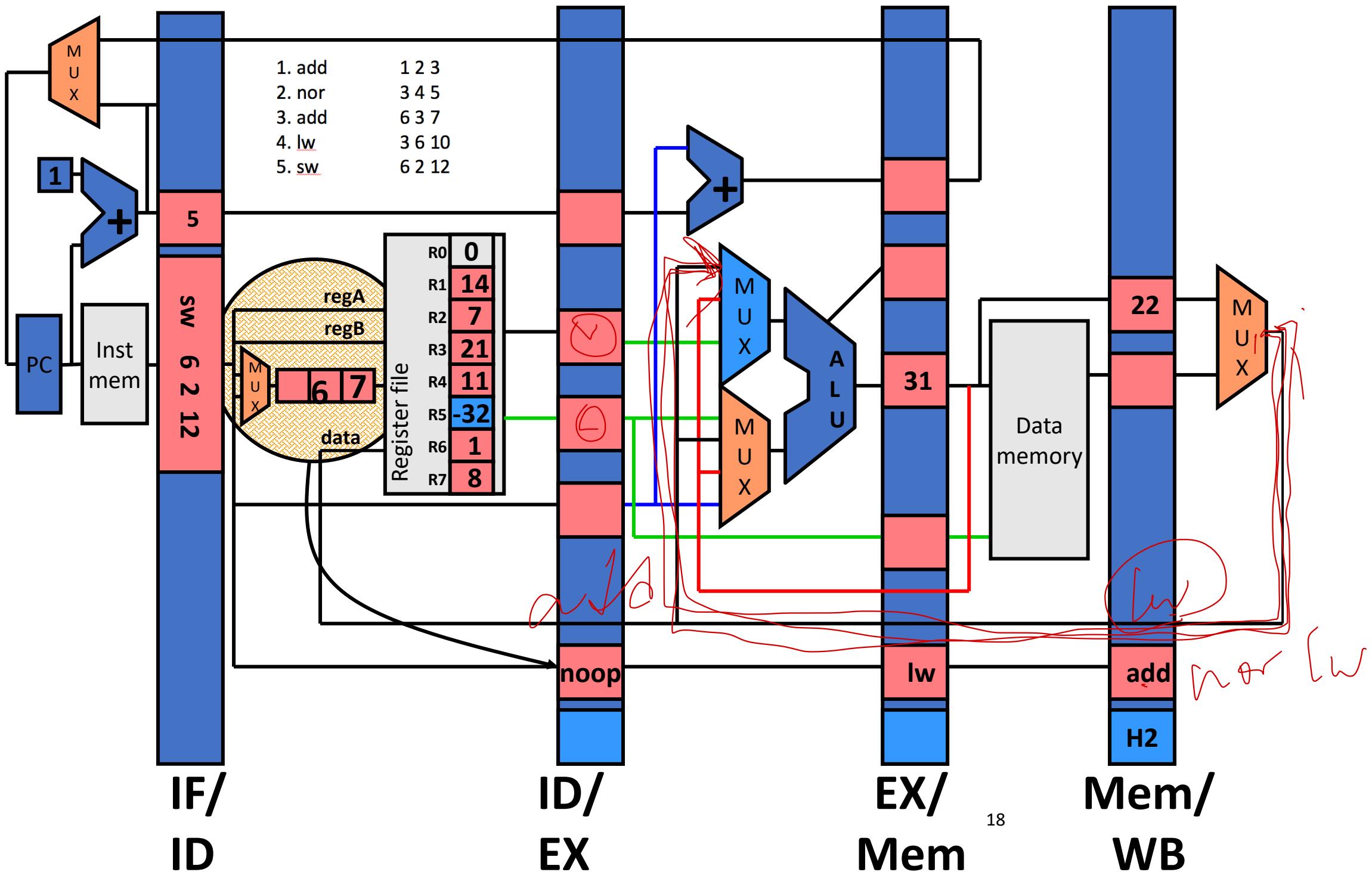
End of cycle 5



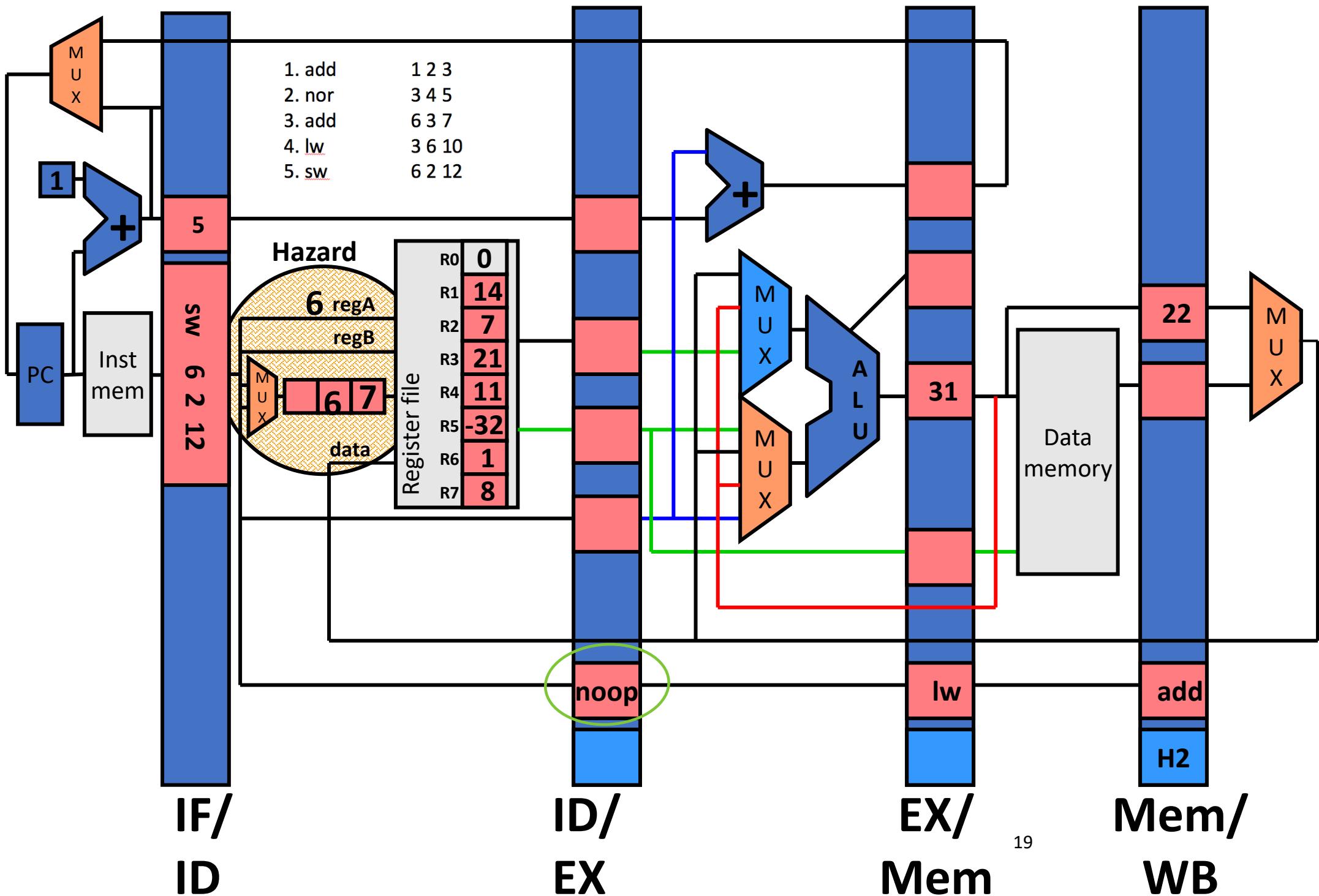
First half of cycle 6



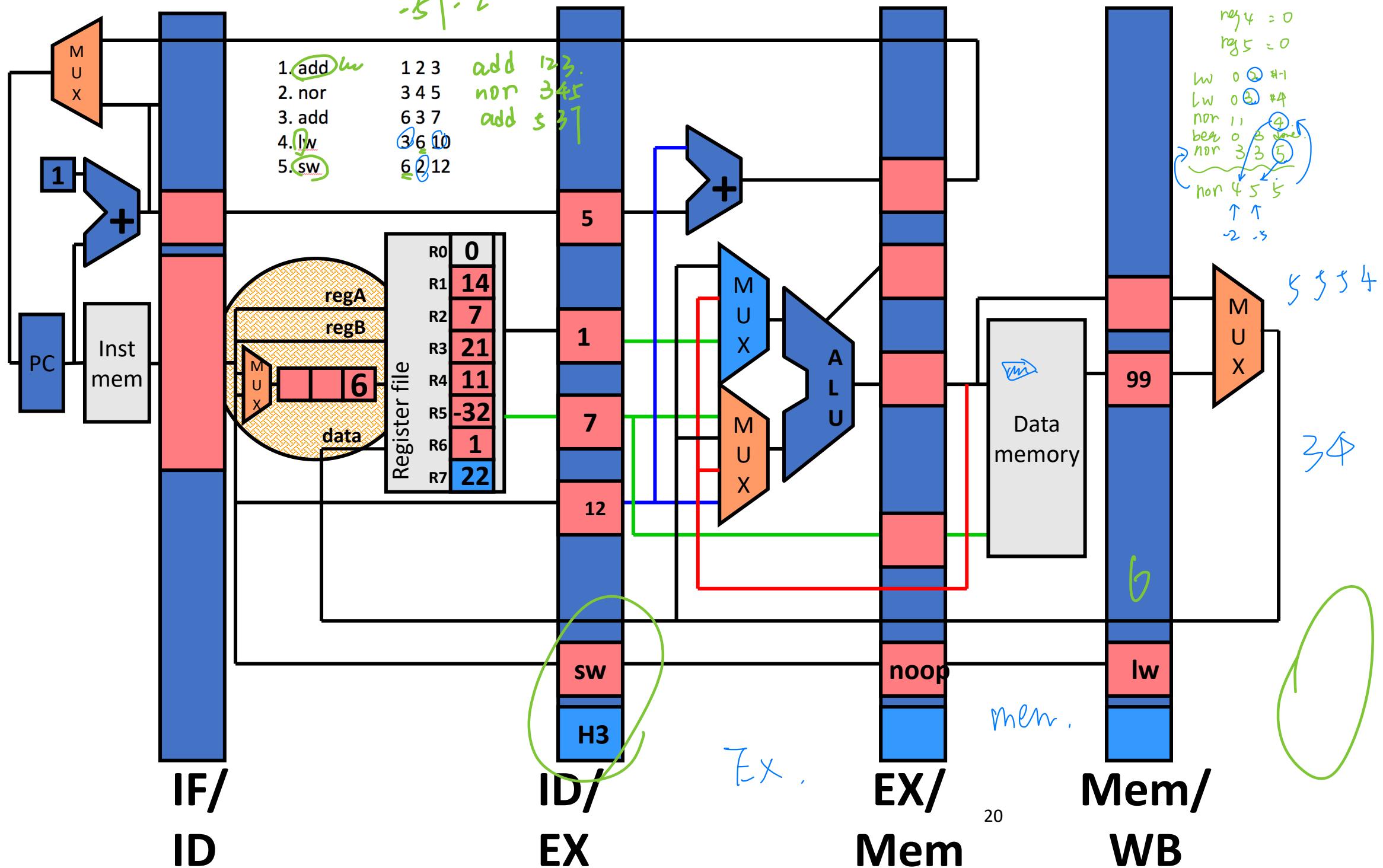
End of cycle 6



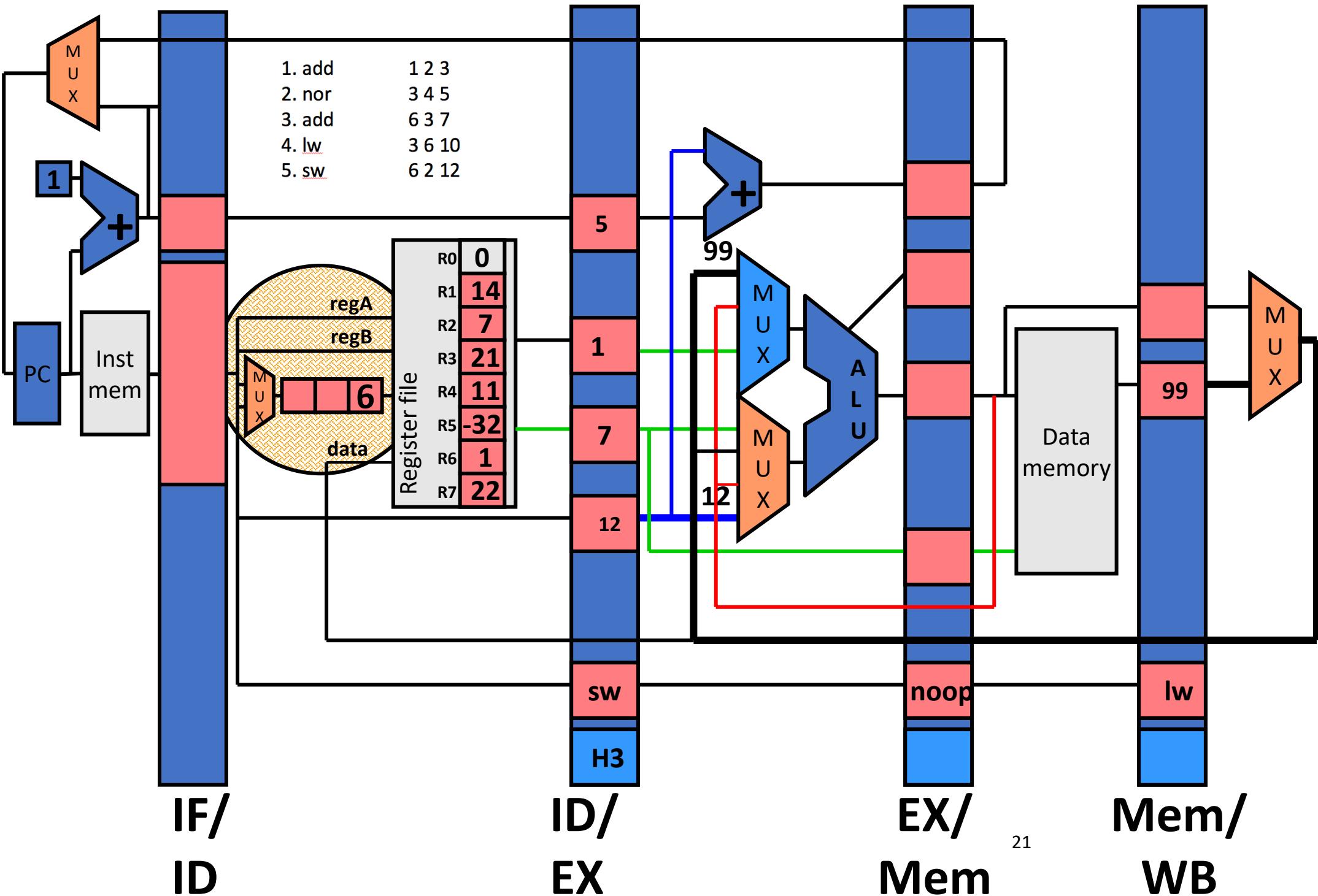
First half of cycle 7



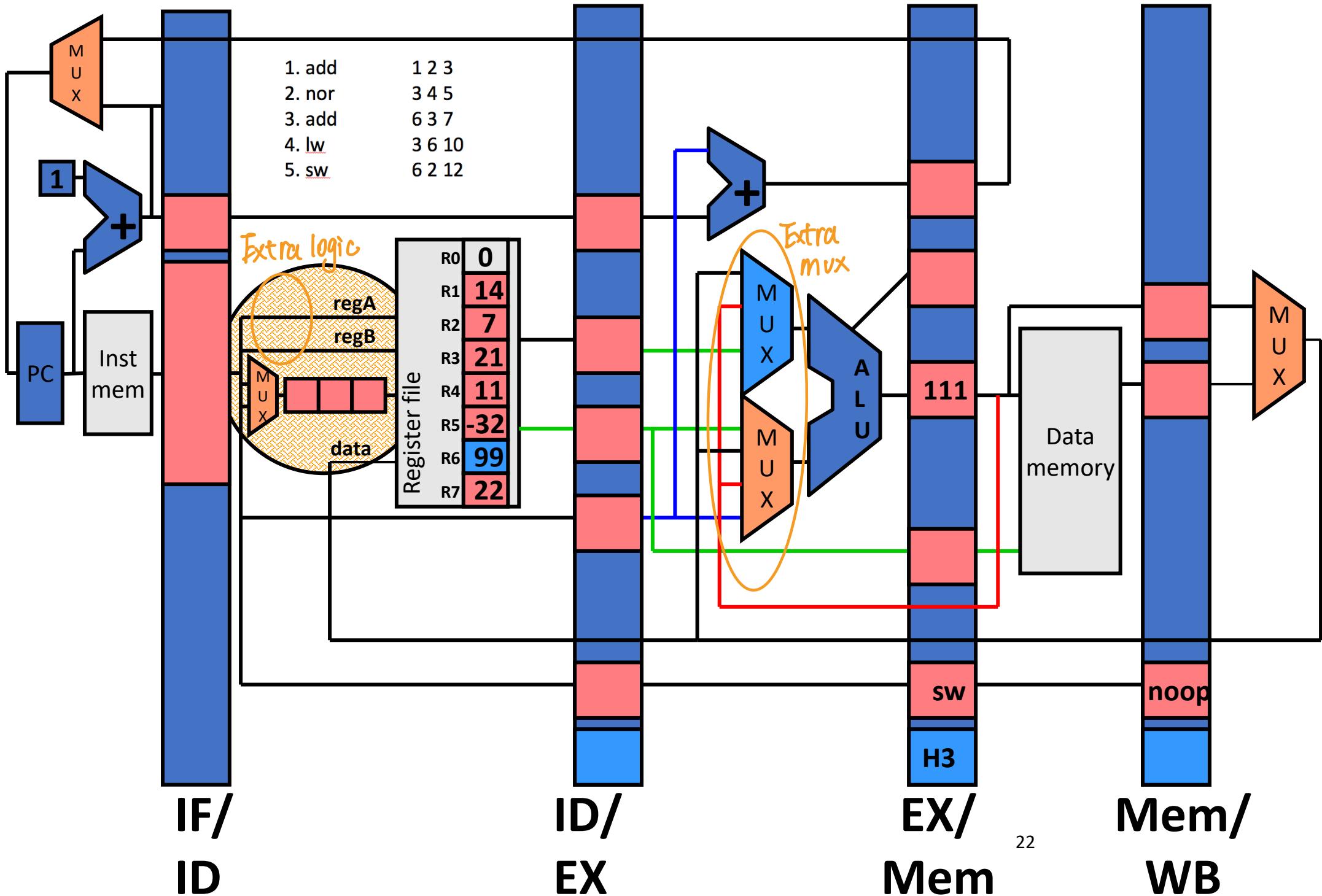
End of cycle 7



First half of cycle 8



End of cycle 8



Time Graph

① 所有的 forward 都不需要特别操作.

② 只有前一个是 "lw" 且紧跟一个 dependent use of that load
immediately afterward.

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID	EX	ME								
add 6 3 7			IF	ID	EX	ME							
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	ID*	ID	EX	ME	WB			

Time Graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID	EX	ME	WB							
add 6 3 7			IF	ID	EX	ME	WB						
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	ID*	ID	EX	ME	WB			

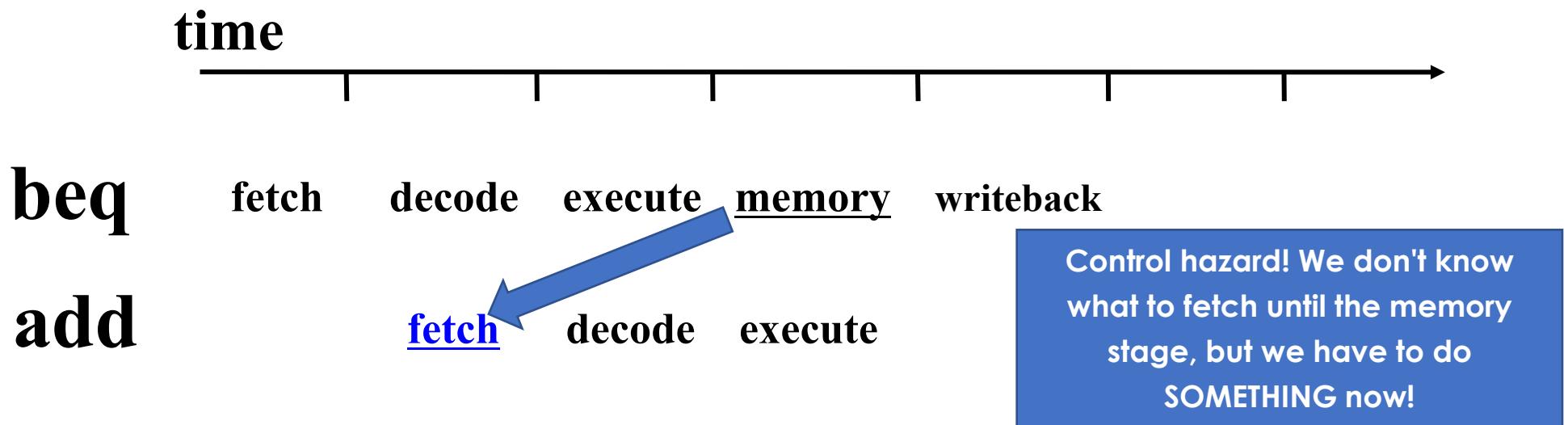
Other issues

- What other instruction(s) have we been ignoring so far??
- Branches!! (Let's not worry about jumps yet)
- Sequence for BEQ:
 - Fetch: read instruction from memory
 - Decode: read source operands from registers
 - Execute: calculate target address and test for equality
 - Memory: Send target to PC if test is equal
 - Writeback: nothing
 - Branch Outcomes
 - Not Taken
 - $PC = PC + 1$
 - Taken
 - $PC = \text{Branch Target Address}$



Control Hazards

beq	1	1	10
add	3	4	5



Approaches to handling control hazards

- 3 strategies – similar to handling data hazards
 - 1. Avoid
 - Make sure there are no hazards in code
 - 2. Detect and stall
 - Delay fetch until branch resolved
 - 3. Speculate and squash-if-wrong
 - Guess outcome of branch
 - Fetch instructions assuming we're right
 - Stop them if they shouldn't have been executed



Avoiding Control Hazards

- Don't have branch instructions!
 - Possible, but not practical
 - ARM offers **predicated** instructions (instructions that throw away result if some condition is not met)
 - Allows replacement of if/else conditions
 - Hard to use for everything
 - Not covered more in this class

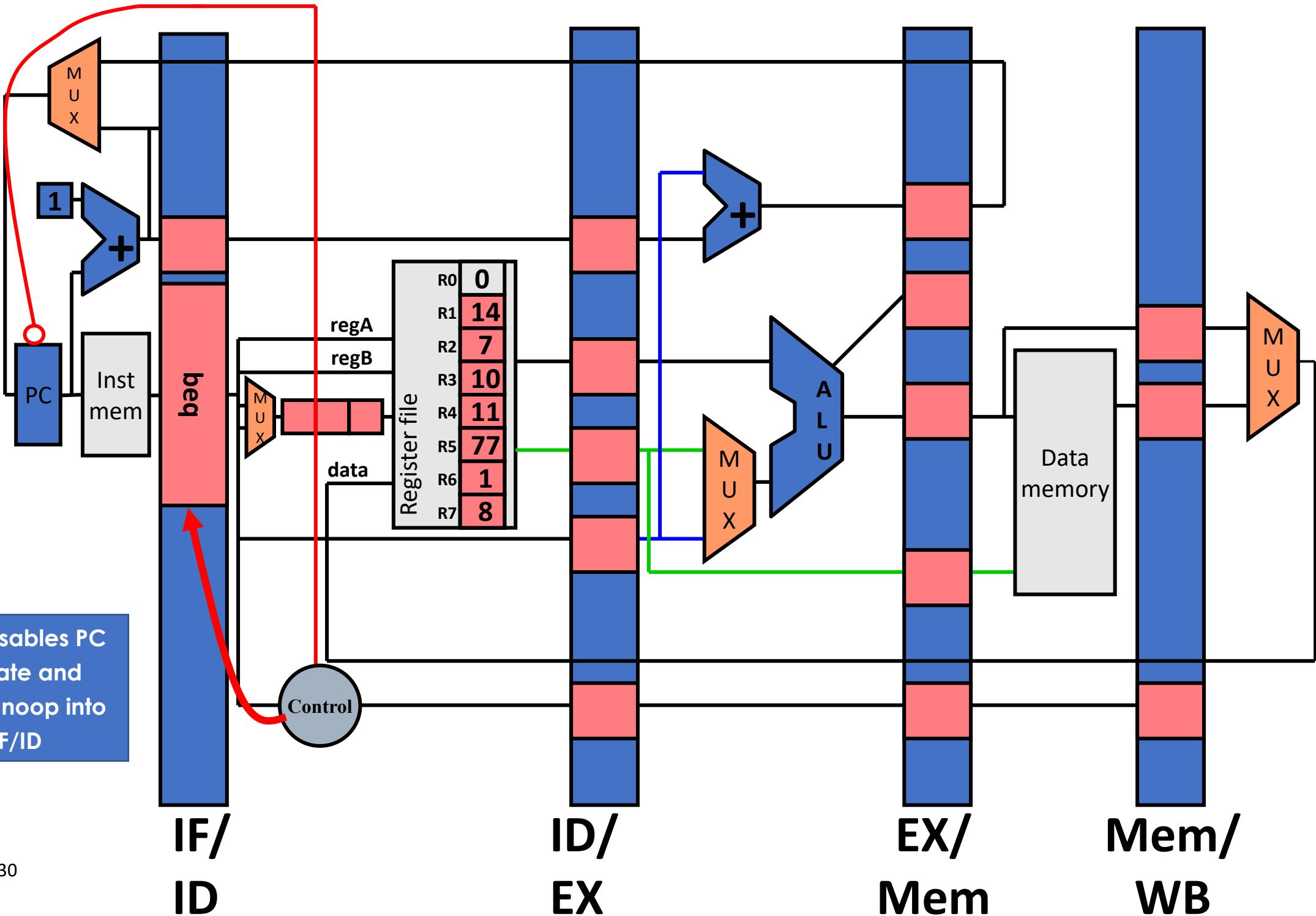
```
if ( condition ) {  
    x++;  
} add. P ↴
```

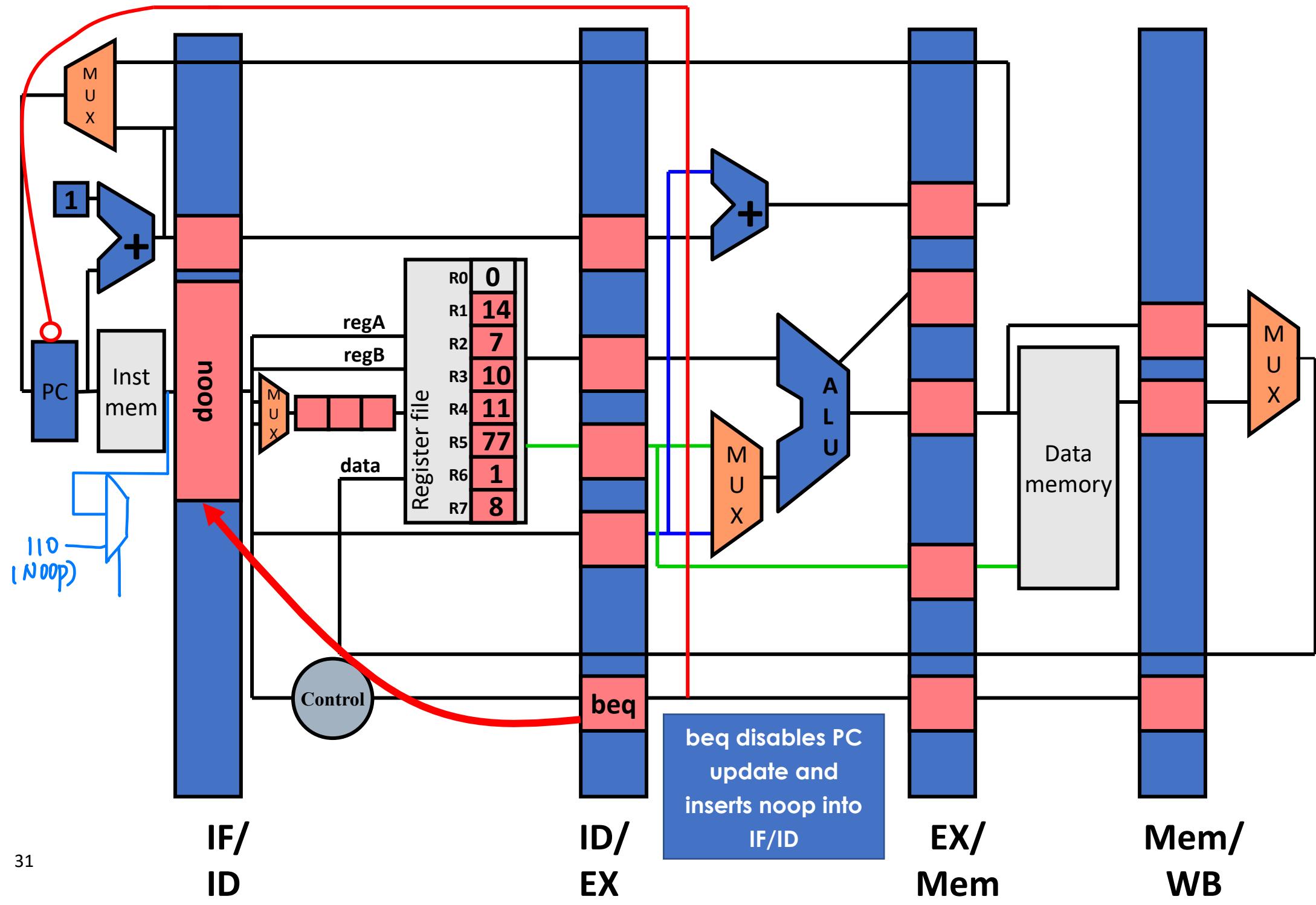
convert the control hazard to data hazard.

Detect and Stall

- Detection
 - Wait until decode
 - Check if opcode == beq or jalr
- Stall
 - Keep current instruction in fetch
 - Insert noops
 - Pass noop to decode stage, not execute!

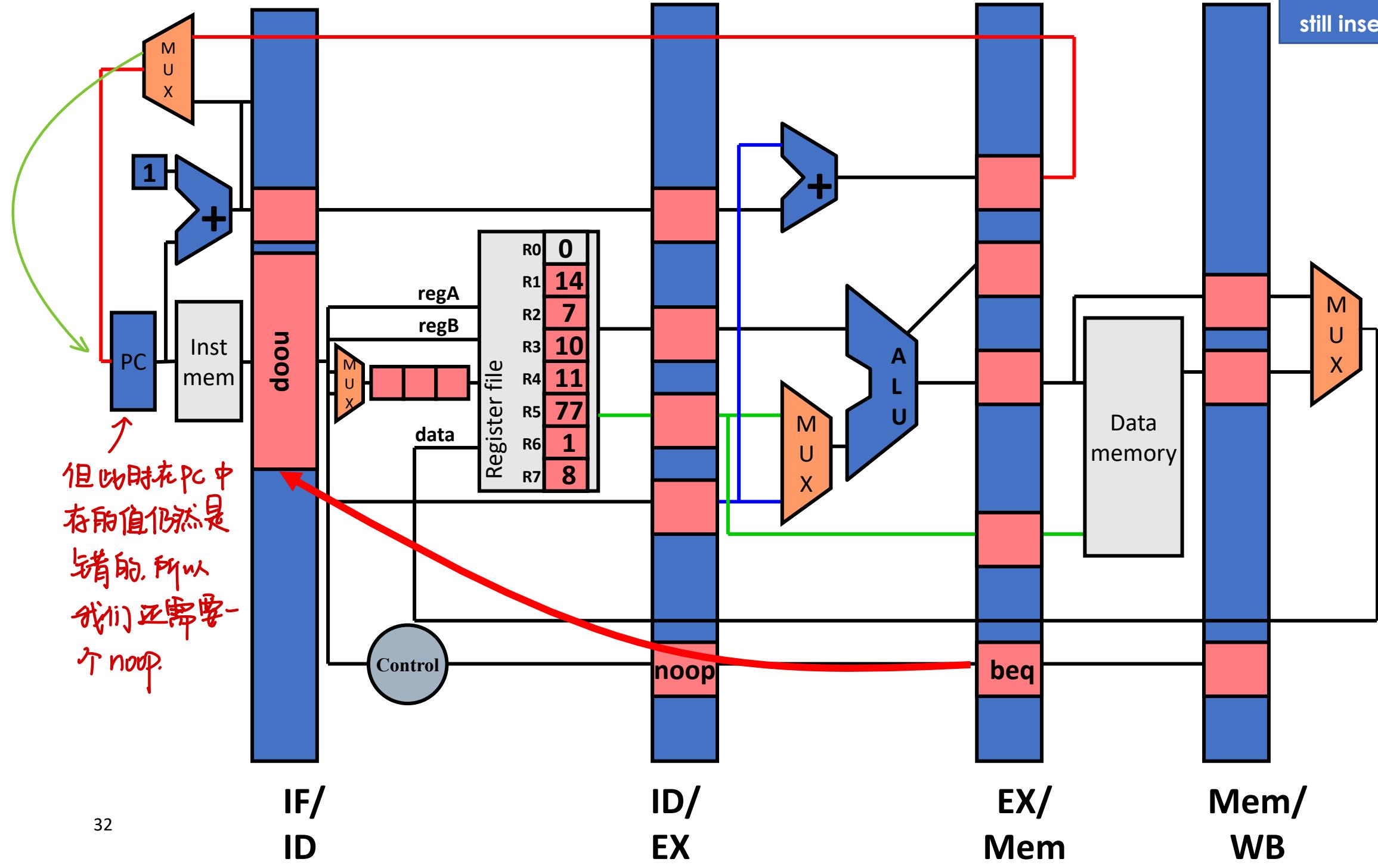




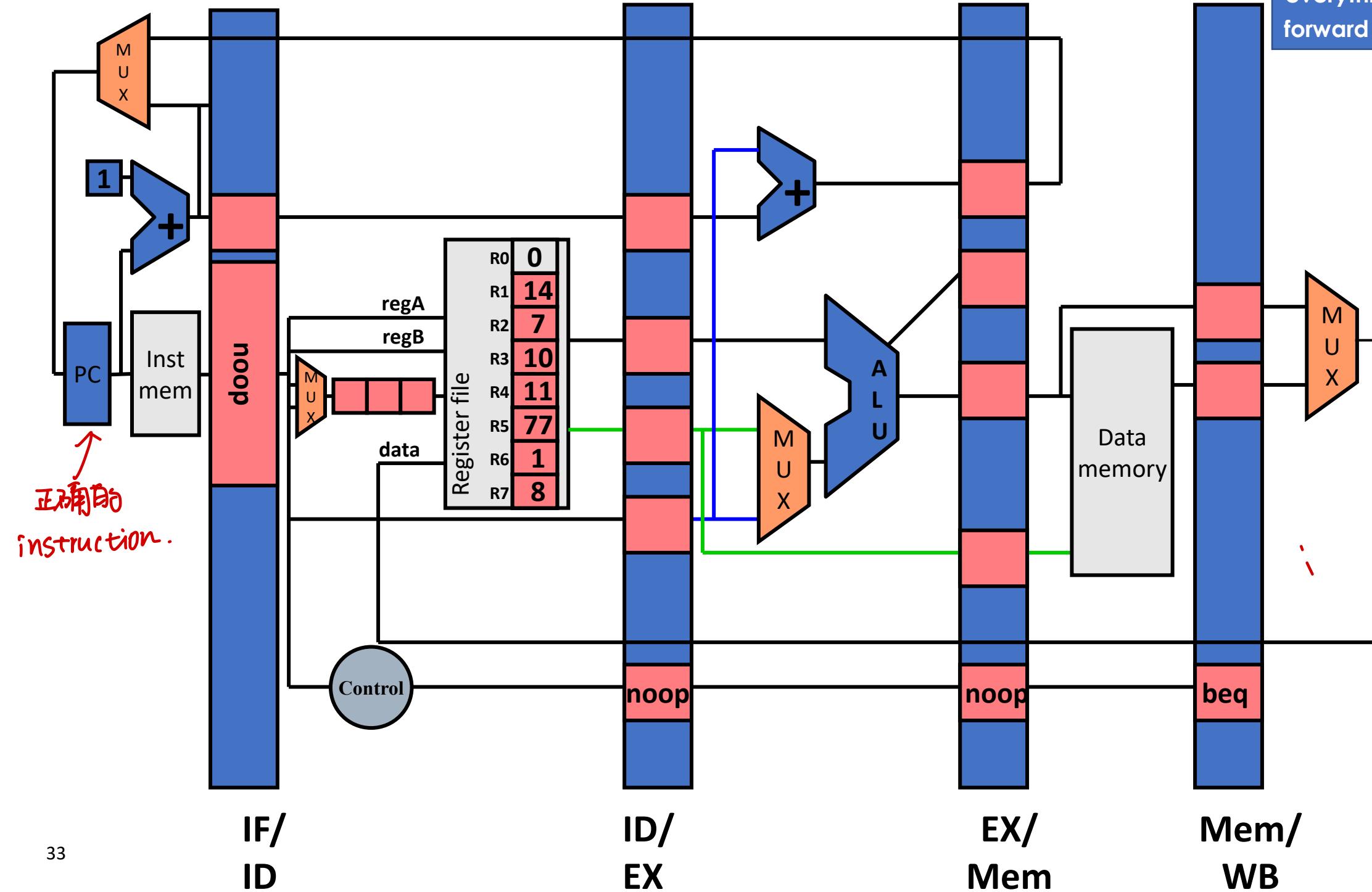


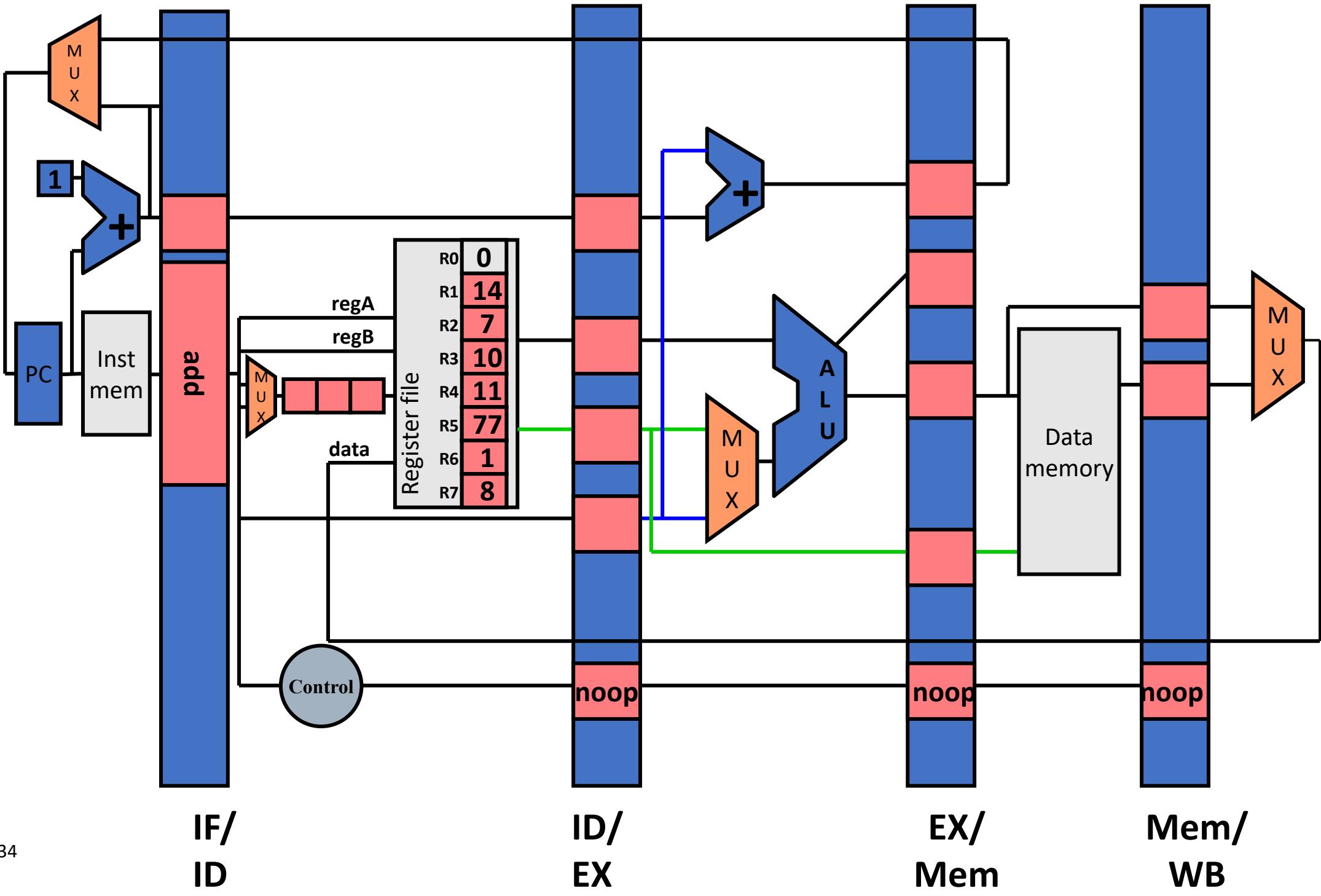
we assume ALU will use most of the cycle time.
so we pass the value from the pipeline register.

beq DOESN'T
disable PC and
sends target, but
still inserts NOOP



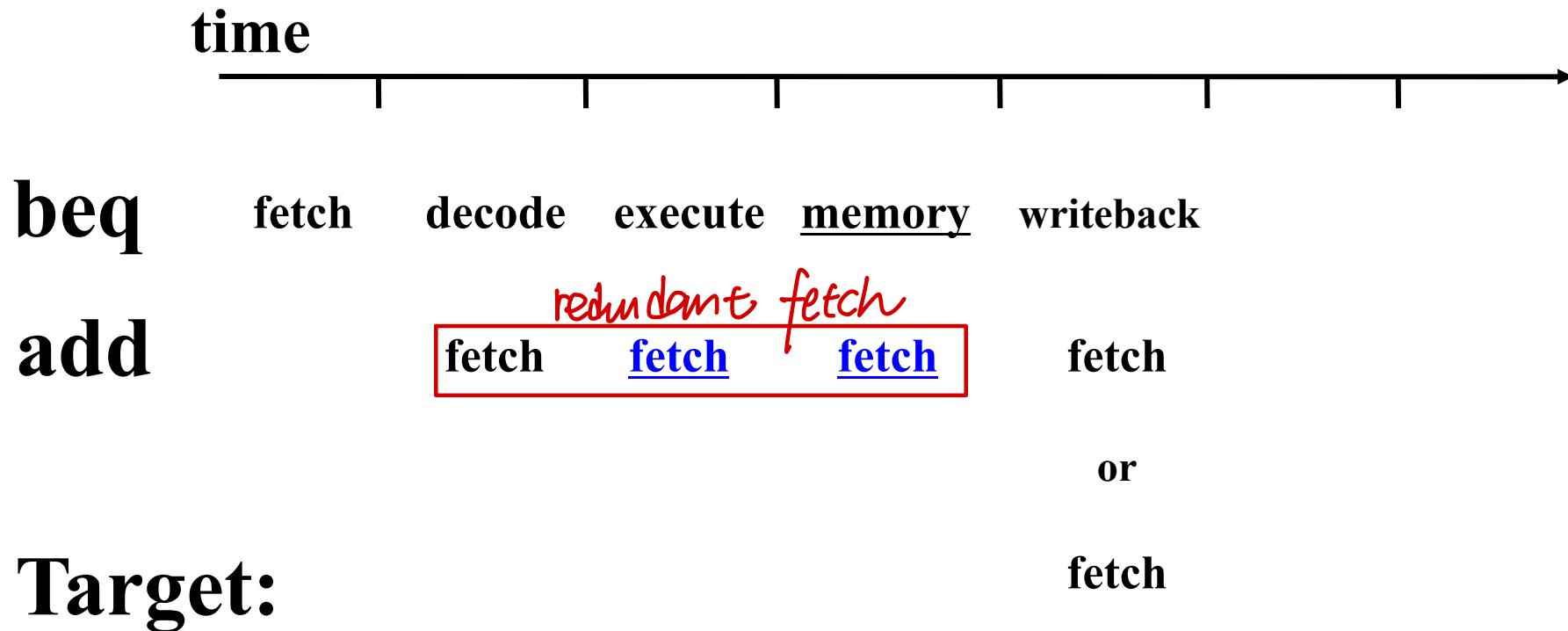
Target PC is now sent to memory, everything moves forward as normal





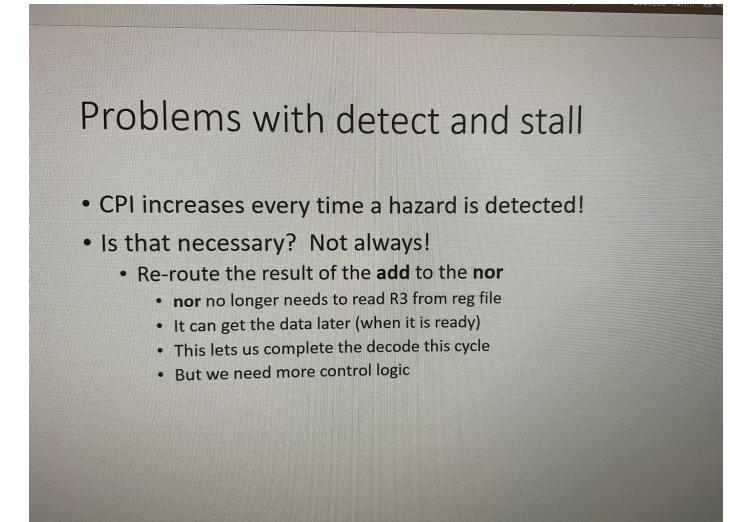
Control Hazards

beq	1	1	10
add	3	4	5



Problems with Detect and Stall

- CPI increases every time a branch is detected!
- Is that necessary? Not always!
 - Branch not always taken
 - Let's assume it is NOT taken...
 - In this case, we can ignore the beq (treat it like a noop)
 - Keep fetching PC + 1
 - What if we're wrong?
 - OK, as long as we do not COMPLETE any instruction we mistakenly execute
 - I.e. DON'T write values to register file or memory



Next time

- Control hazards
 - How do branches work with pipelining?

Extra Slides

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- **Speculate-and-Squash**
- Exceptions
- Practice Performance Problems
 - Problem 1
 - Problem 2
 - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

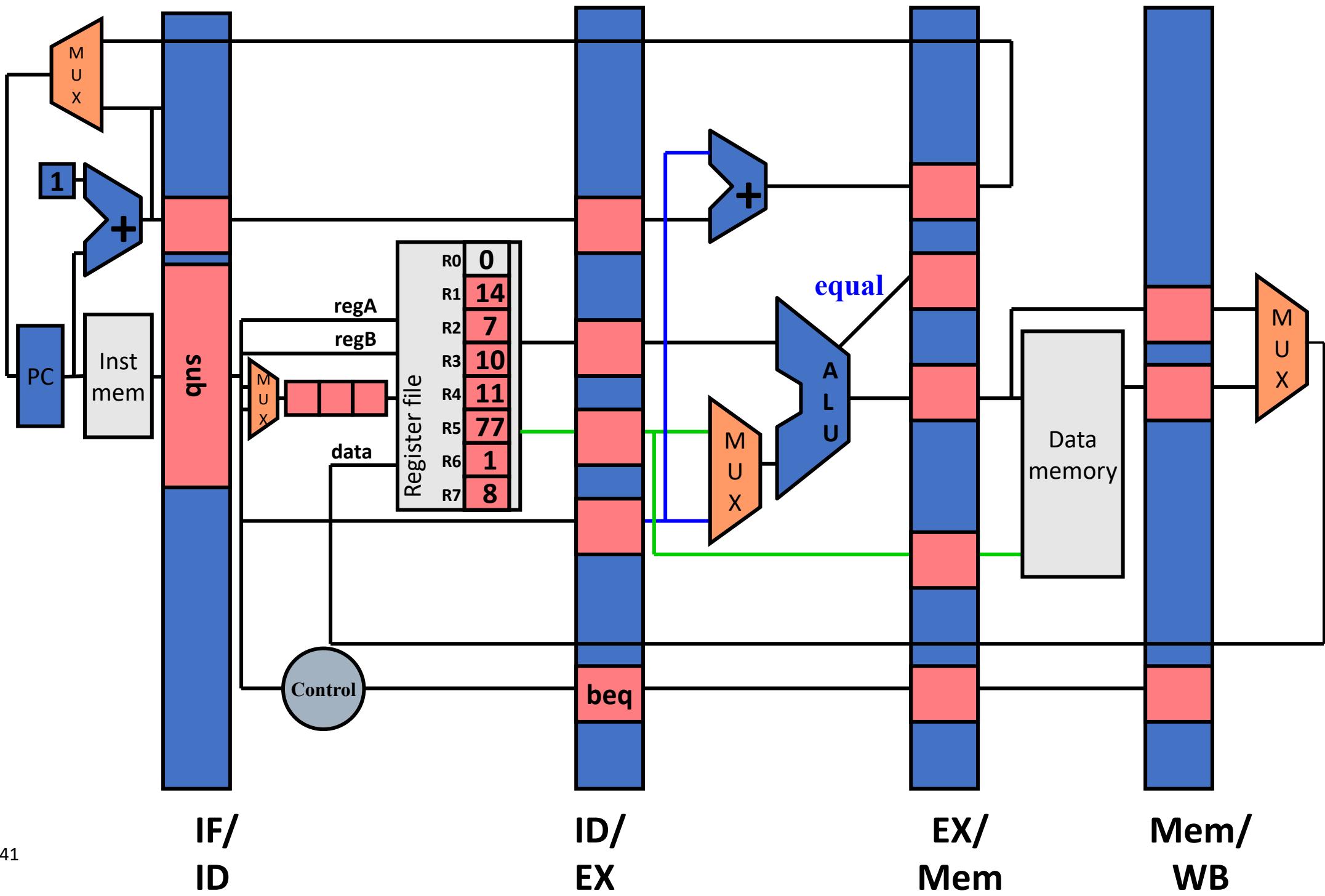


Speculate and Squash

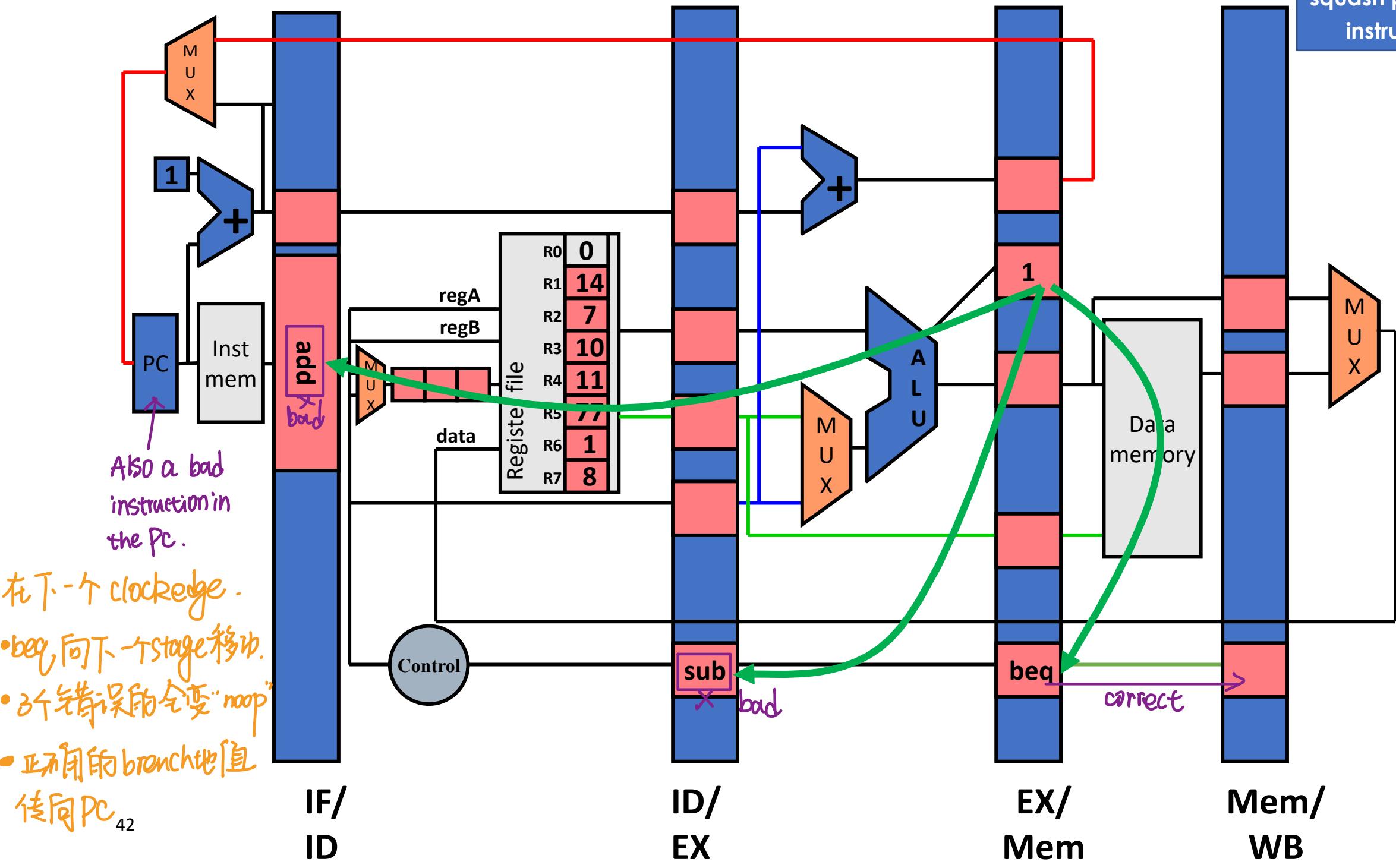
- Speculate: assume not equal
 - Keep fetching from PC+1 until we know that the branch is really taken
- Squash: stop bad instructions if taken *if we speculate incorrectly, then overwrite all of the bad instruction with "noop", then send the correct target address to the program counter.*
 - Send a noop to Decode, Execute, and Memory
 - Sent target address to PC

Resolve branch in
execute stage

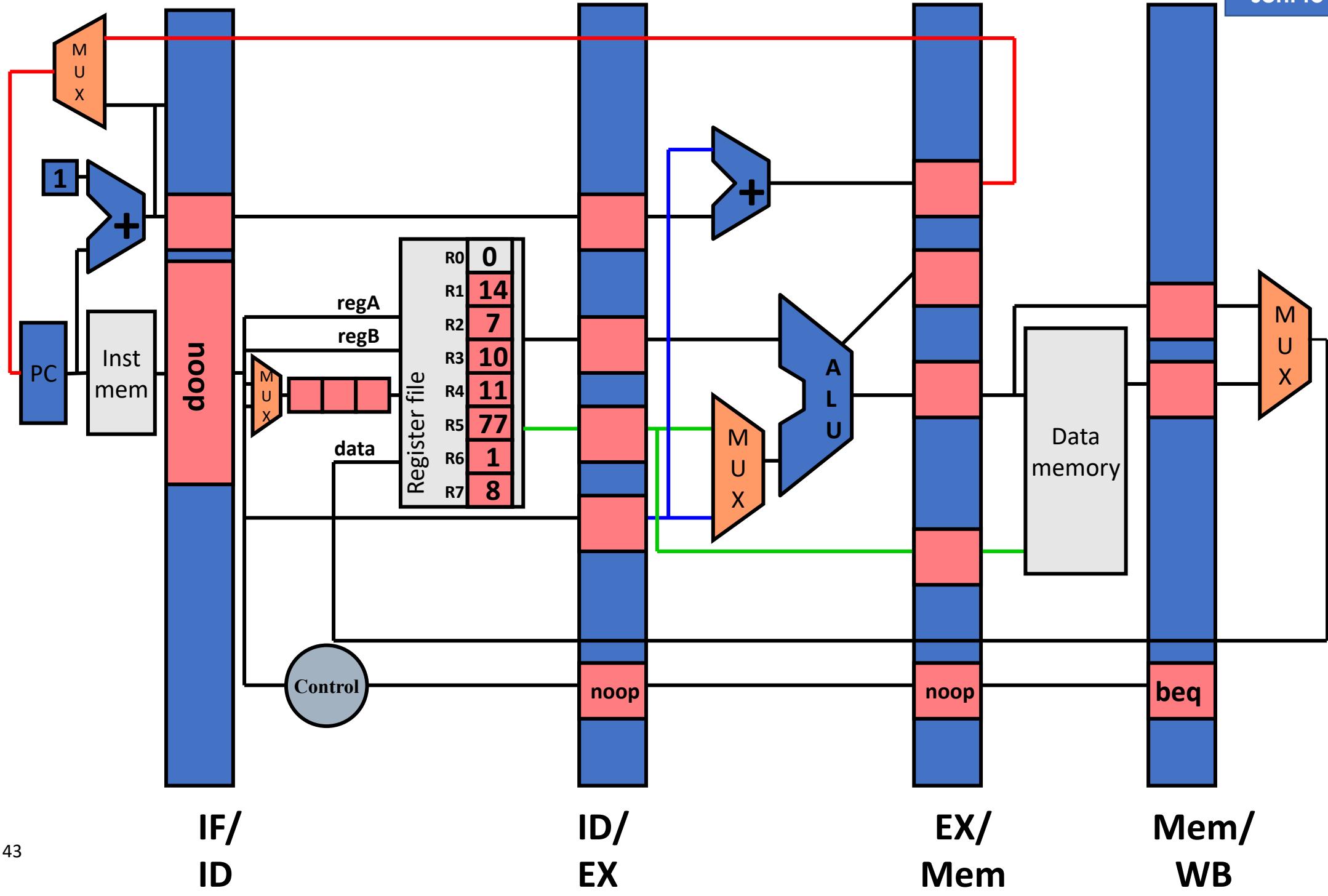
wait until the begining of the next cycle



Send target to PC
in Mem stage,
squash previous 3
instructions



3 noops inserted,
correct PC being
sent to memory



Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- **Exceptions**
- Practice Performance Problems
 - Problem 1
 - Problem 2
 - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor



What can go wrong?

- ❑ **Data hazards**: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if is about to be written.
- ❑ **Control hazards**: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- ❑ **Exceptions**: How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Exceptions

- ❑ Exception: when something unexpected happens during program execution.
 - Example: divide by zero.
 - The situation is more complex than the hardware can handle
 - So the hardware branches to a function, an “exception handler” which is code to try to deal with the problem.
- ❑ The exact way to set up such an exception handler will vary by ISA.
 - With C on x86 you would use <signal.h> functions to handle the “SIGFPE” signal.
 - There is a pretty good [Hackaday article](#) on this if you want to learn more.



Exceptions and Pipelining

- The hardware branches to the “exception handler”
 - This means that any instruction which can “throw” an exception *could* be a branch.
 - Throwing an exception should be rare (“exceptional”)

- So we would treat it much like a branch we predicted as “not taken”
 - Squash instructions behind it and then branch
 - It will introduce stalls, but since it should be rare, we don’t worry about it.
 - “Make the common case fast”.



Extra Slides



Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- **Practice Performance Problems**
 - **Problem 1**
 - Problem 2
 - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor



Classic performance problem

- ❑ Program with following instruction breakdown:

Iw	10%	$CPI = 1 + 0.1 \times 0.2 \times 1 + 0.25 \times 0.2 \times 3$
sw	15%	$= 1 + 0.02 + 0.15$
beq	25%	
R-type	50%	$= 1.17$

- ❑ Speculate “always not-taken” and squash. 80% of branches not-taken
- ❑ Full forwarding to execute stage. 20% of loads stall for 1 cycle
- ❑ What is the CPI of the program?
- ❑ What is the total execution time per instruction if clock frequency is 100MHz?

$$1 + 0.1 \times 0.2 \times 1 + 0.25 \times 0.2 \times 3$$

$$CPI = 1.17$$

$$\text{Time} = CPI \times \frac{1}{100\text{MHz}} = 1.17 \times \frac{1}{1 \times 10^2 \times 10^6} = 1.17 \times 10^{-9} \text{s} = 1.17 \text{ ns}$$

Classic performance problem

- ❑ Program with following instruction breakdown:

Iw 10%

sw 15%

beq 25%

R-type 50%

- ❑ Speculate “always not-taken” and squash. 80% of branches not-taken

- ❑ Full forwarding to execute stage. 20% of loads stall for 1 cycle

- ❑ What is the CPI of the program?

- ❑ What is the total execution time per instruction if clock frequency is 100MHz?

$$\text{CPI} = 1 + 0.10 \text{ (loads)} * 0.20 \text{ (load use stall)} * 1 \frac{10^{-9} \text{ ns.}}{\text{instruction}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{time}}{\text{cycle}} \cdot \frac{\text{instruction}}{\text{program}}$$

$= 100 \times 10^6 = 10^8 \rightarrow \text{inverse } 10^{-8}$

$\text{So } 10^{-9} = 10 \text{ ns.}$

$$\text{CPI} = 1 + 0.02 + 0.15 = 1.17$$

$$\text{Time} = 1.17 * 10 \text{ ns} = 11.7 \text{ ns per instruction}$$

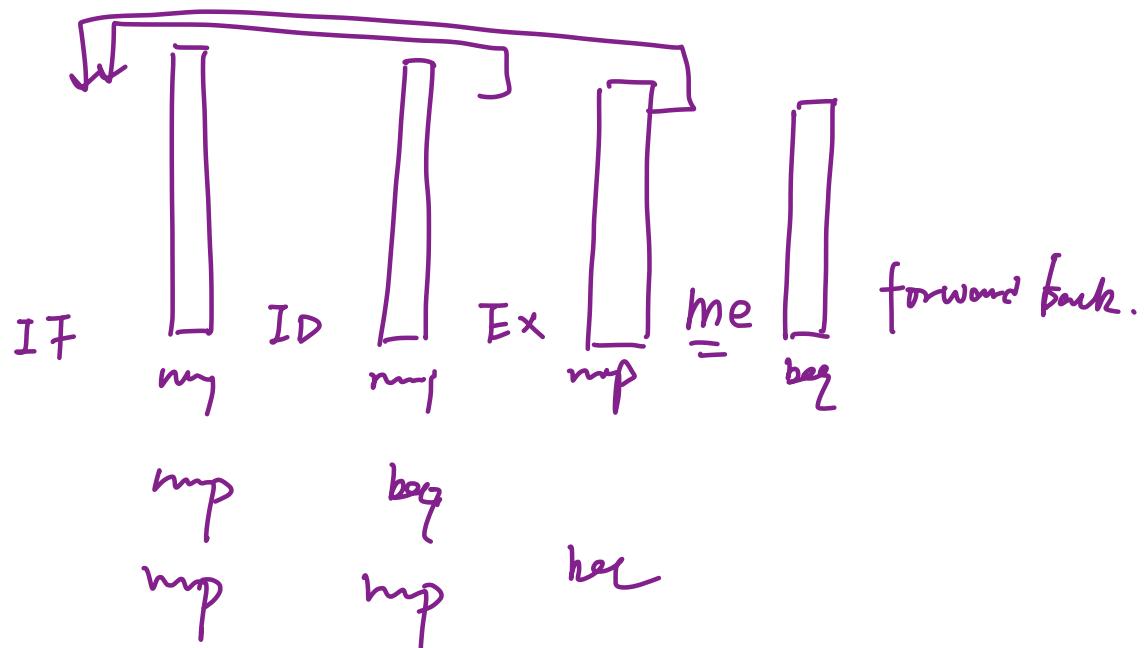
Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
 - Problem 1
 - **Problem 2**
 - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor



Classic performance problem (cont.)

- ❑ Assume branches are resolved at Execute?
 - What is the CPI?
 - What happens to cycle time?



Classic performance problem (cont.)

- ❑ Assume branches are resolved at Execute?
 - What is the CPI?
 - What happens to cycle time?

$$\text{CPI} = 1 + 0.10 \text{ (loads)} * 0.20 \text{ (load use stall)} * 1$$

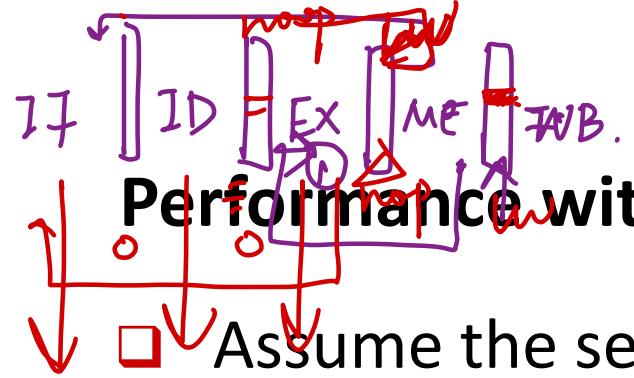
$$+ 0.25 \text{ (branch)} * 0.20 \text{ (miss rate)} * 2$$

$$\text{CPI} = 1 + 0.02 + 0.1 = 1.12$$

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
 - Problem 1
 - Problem 2
 - **Problem 3**
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor





Performance with deeper pipelines

Assume the setup of the previous problem.

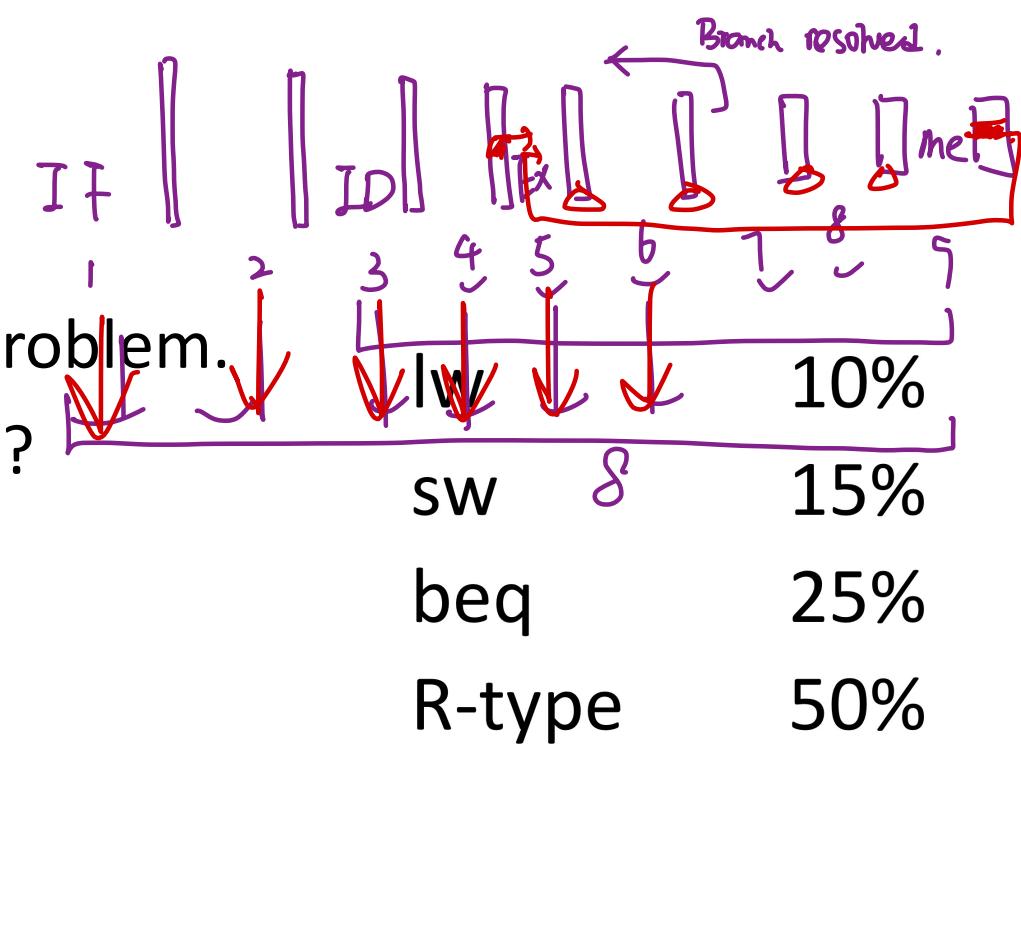
What if we have a 10 stage pipeline?

- Instructions are fetched at stage 1.
- Register file is read at stage 3.
- Execution begins at stage 5.
- Branches are resolved at stage 7.
- Memory access is complete in stage 9.

What's the CPI of the program? $1 + 0.1 \times 0.2 \times (9-5) + 0.25 \times 0.2 \times 6$

If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

$$1 + 0.1 \times 0.2 \times \cancel{5}^4 + 0.25 \times 0.2 \times 6 = 1.38$$



Performance with deeper pipelines

- ❑ Assume the setup of the previous problem.
- ❑ What if we have a 10 stage pipeline?
 - Instructions are fetched at stage 1.
 - Register file is read at stage 3.
 - Execution begins at stage 5.
 - Branches are resolved at stage 7.
 - Memory access is complete in stage 9.
- ❑ What's the CPI of the program?
- ❑ If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

$$\text{CPI} = 1 + 0.10 \text{ (loads)} * 0.20 \text{ (load use stall)} * 4 + 0.25 \text{ (branch)} * 0.20 \text{ (N stalls)} * 6$$

$$\text{CPI} = 1 + 0.08 + 0.30 = 1.38$$

$$\text{Time} = 1.38 * 5\text{ns} = 6.9 \text{ ns per instruction}$$

